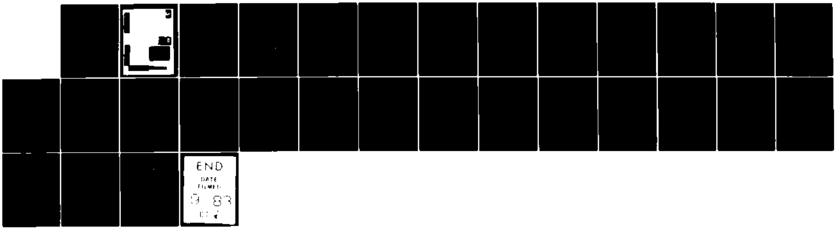
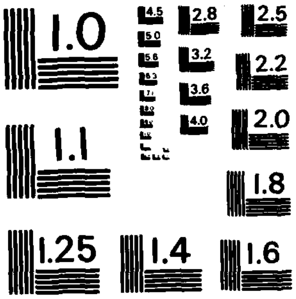
AD A132 512 SCHEDULING SUPERCOMPUTERS(U) MINNESOTA UNIV MINNEAPOLIS 1/1
DEPT OF COMPUTER SCIENCE S SAHNI FEB 83 TR-83-3
N00014-80-C-0650

UNCLASSIFIED F/G 12/1 NL



END
DATE
FILMED
9 83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Scheduling Supercomputers

by

Sartaj Sahni

Technical Report 83-3

February 1983

Computer Science Department

Institute of Technology

136 Lind Hall

University of Minnesota

Minneapolis, Minnesota 55455

Scheduling Supercomputers

by

Sartaj Sahni

Technical Report 83-3

February 1983

Scheduling Supercomputers*

Sartaj Sahni

University of Minnesota

Abstract

We develop good heuristics to schedule tasks on supercomputers. Supercomputers comprised of multiple pipelines as well as those comprised of asynchronous multiple processors are considered. In addition, we consider the case when different pipes or processors run at different speeds.

Keywords and Phrases

Supercomputers, pipelines, asynchronous processors, scheduling, heuristics.

* This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650 and in part by the Microelectronics and Information Sciences Center at the University of Minnesota.

1. Introduction

A block diagram for a multiple pipeline vector supercomputer ([6]) is given in Figure 1. Instruction fetches and decodes are carried out by the *instruction processing unit*. Scalar instructions are sent to the *scalar processor* while vector instructions are sent to the *vector controller*. The vector controller receives vector instructions from the instruction processing unit. These instructions are set up on the vec ,or access controller, buffer and pipeline. Data is brought to and from the pipelines by the *vector access controller* via the vector buffer. The *vector buffer* is essentialy a cache that is used to close the gap between memory access speed and vector pipeline speed. The *vector pipeline* actually consists of several (say m) independent pipelines. Each pipeline is capable of executing every instruction (though during a single vector instruction the instruction executed does not change) and the vector controller is capable of scheduling several vector instructions simultaneously.

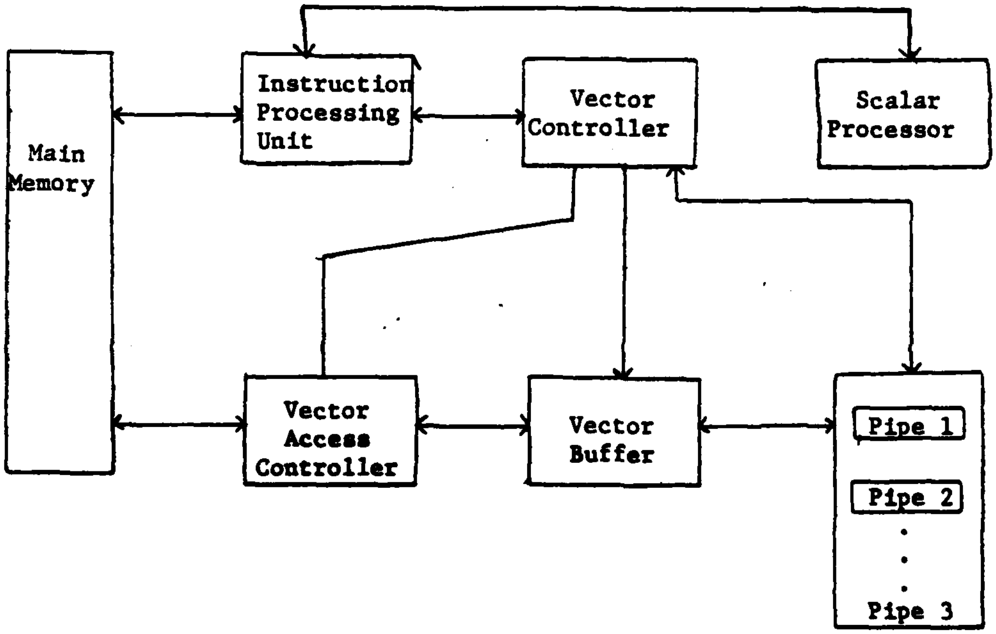


Figure 1 Block diagram of a multi pipeline vector supercomputer.

The pipelines constituting the vector pipeline may be identical or uniform. Thus with pipeline i we may associate a speed s_i, $1 \leq i \leq m$. When all the s_is are the same, we say that the pipelines are *identical*. The speed of a pipeline is measured relative to that of a *unit pipeline* which by definition has a speed of 1.

There are three aspects to executing a vector task on a pipeline. First, there is the time needed to set up the instruction and get the first operand pair to the pipeline. This is the *start up* time. Next, there is the time needed to perform the instruction on an operand pair and bring in the next operand pair. This is the *latency time*. Finally, there is the *flush time*. This is the time needed to perform the instruction on the last operand pair and move the results out of the pipeline. In this paper, we shall make the simplifying assumption that the start up, latency, and flush times on a unit pipeline are the same for every vector instruction. Let t_0' denote the sum of the start up and flush times and let t_l denote the latency time. The total time, t, needed (called the *processing time*) by a unit pipeline to run a vector instruction on a vector of length L is given by the equation [6]:

$$
\begin{aligned}
t &= t_0' + t_l *(L - 1) \\
&= (t_0' - t_l) + t_l L) \\
&= t_0 + t_l L
\end{aligned}
$$

where $t_0 = t_0' - t_l$ is called the *overhead time*.

For a typical unit pipeline, t_0 will be much larger than t_l. A pipeline with speed s can in δ time perform $s*\delta$ units of processing. Thus if a task needs t units of processing on a unit pipeline, it can be completed in t/s time units on a pipeline of speed s.

Let us assume that a set of n tasks is to be scheduled on the m pipelines. In general there will be a precedence relation associated with the task set. However, in this paper we shall consider only the case when this relation is null. I.e., the tasks are independent. Let L_i be the length of the vector task i and let $t_i = t_l * L_i$. We shall require that $t_i > 0$. A unit pipeline will require $t_0 + t_i$ time to complete task i.

A *schedule* is an assignment of tasks (or portions of tasks) to time slots on the pipelines such that:

1. No pipeline executes more than one task at any given time.
2. No task is being executed on more than one pipeline at any time.
3. All tasks are completed by the end of the schedule. Note that tasks may be scheduled preemptively and that every time a task is started, the overhead penalty of t_0 units of processing is incurred. Consequently, tasks (or portions thereof) must not be scheduled for time slots of size less than t_0 / s_i on processor i. The scheduled slot should actually be larger than this if any useful work is to be performed.

The *length* of a schedule is the earliest time by which all the pipelines have completed the work assigned to them. In a *nonpreemptive* schedule, a task is executed continuously from start to finish on the same pipeline. A task is said to be scheduled preemptively if it is assigned to two or more noncontiguous time slots on the same pipeline or is assigned for processing to two or more pipelines. Throughout this paper, we assume that the number of tasks, n, to be scheduled is no less than the number of pipelines, m, available.

The advantages to be reaped from preemptive schedules can be seen from a simple example. Let n = 3, m = 2 , $s_1 = s_2 = 1$, $t_0 = 1$, $t_1 = t_2 = t_3 = 100$. If no preemptions are used, it takes the 2 pipelines 202 time units to complete the 3 tasks (Figure 2a). On the other hand, by using preemptions, the 3 tasks can be completed in 152 time units (Figure 2b). The shaded area in each figure indicates the overhead time of t_0.

We are interested, in this paper, in developing algorithms to schedule task sets so as to minimize the schedule length. Before discussing work previously done on this problem, we introduce another supercomputer model for which this scheduling problem is of interest. Figure 3 gives the block diagram for a supercomputer comprised of m asynchronous and independent processors. Each processor starts with its schedule of tasks (and subtasks) and repeatedly performs the following steps:

1. Set up the next task (or subtask) to be performed. This will involve getting the program and data for this task from the common memory and transfering it to the local memory.

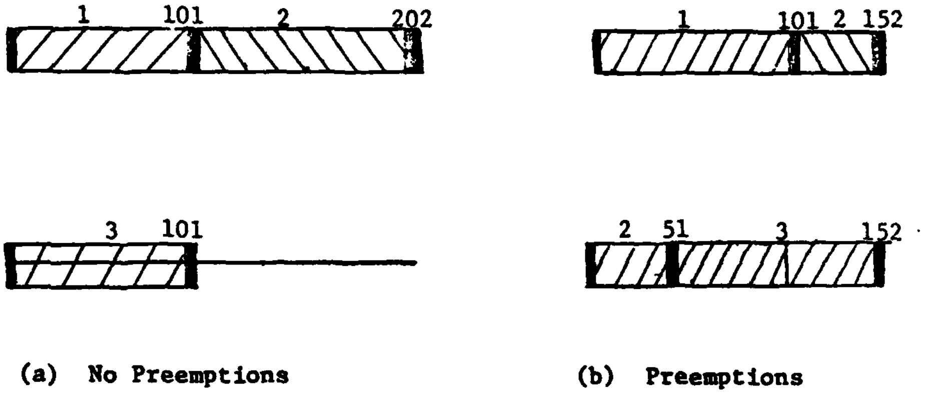


Figure 2 Example schedules

2. Execute the task for the specified duration.
3. Flush the processor. This wculd involve moving the results of the compution back to the common memory.

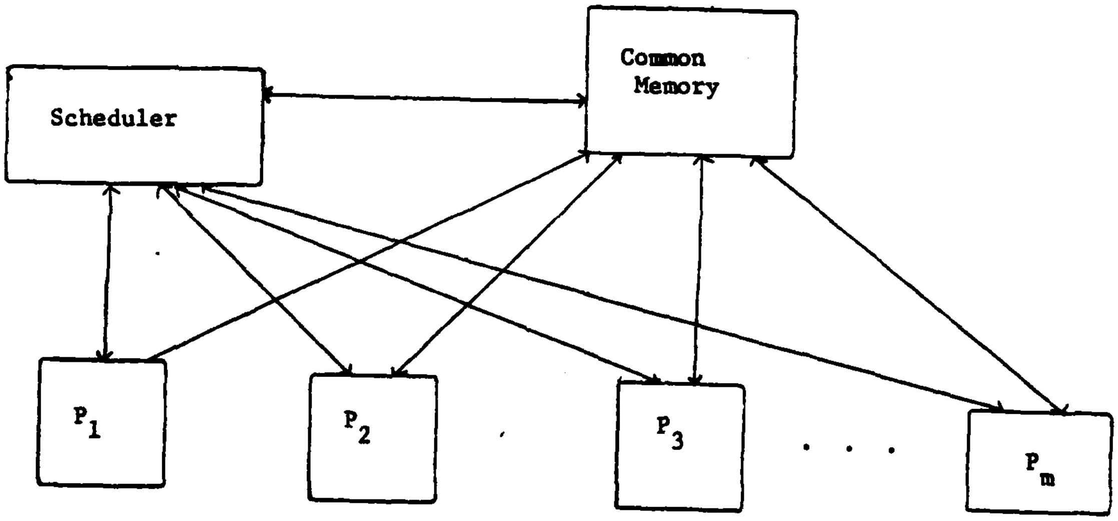


Figure 3 Block diagram of a supercomputer with asynchronous processors.

As in the case of a vector pipeline supercomputer, the m independent processors may or may not be identical. In general, there will be a speed s_i and a local memory size μ_i associated with processor i. Task i will require a total of t_i units of processing (excluding overhead) on a unit processor (i.e., a processor whose speed is 1). In addition, task i will require u_i units of local memory to run. Hence, this task (or portions of it) can be run only on those processors that have at least u_i units of local memory. Once again, we are interested in constructing schedules that have minimum length. We make the simplifying assumption that the common memory is sufficiently interleaved that all processors can do their set up and flush simultaneously.

It is not too difficult to see that the problem of constructing minimal length preemptive schedules for an m pipeline vector supercomputer is identical to that of constructing such schedules for a supercomputer that has m asynchronous processors all of which have the same amount of local memory. So, in future discussion we shall explicitly refer only to the m asynchronous processor case. All our results trivially carry over to the case of m pipelines.

It is well known that constructing minimum length preemptive schedules is NP-hard even when there are only 2 identical processors with equal memory size ([4]). When the start up and flush time is zero (i.e., $t_0 = 0$), optimal schedules may be constructed efficiently. McNaughton [10] has developed an O(n) algorithm for the case when all processors have the same speed as well as the same memory capacities. The algorithm developed by Kafura and Shen [5] for the case when all processors have the same speed but have different memory sizes oan be easily implemented to run in O(nlogm) time. Gonzalez and Sahni [3] have developed an O(n + mlogm) algorithm for the case of uniform processors having the same memory size. The general problem of uniform processors with different memory sizes has been considered by Lai and Sahni [7] and by Martel [9].

[1], [2], [8], [11], and [12] are some other references on work related to the scheduling of multi pipelined supercomputers.

As stated above, the problem we are considering in this paper (i.e., construct minimum length schedules) is NP-hard. Hence, it is extremely unlikely that there exist efficient (i.e., polynomial time) algorithms that solve our problem. We shall therefore relax the requirement that the schedules constructed be minimal and only require that the schedules be constructed quickly and be "good".

Su and Hwang [12] have developed an efficient algorithm, SU, to schedule n tasks on m identical processors with the same memory size. Their algorithm runs in O(n) time and generates solutions that are quite good. Specifically, if we let t_a, w_0 and w_{SU} be as below:

$$t_a = \max \{ \max_i \{ t_i + t_0 \}, \sum_i (t_i + t_0)/m \}$$

w_0 = length of minimum length schedule

w_{SU} = length of schedule generated by the Su-Hwang algorithm

then,

$$\begin{aligned} w_{SU} &\le t_a + (m-1)t_0/2 \\ &\le w_0 + (m-1)t_0/2 \end{aligned}$$

Using algorithm SU, Su and Hwang, further showed how a task set with tree precedence could be scheduled such that the schedule length was no more than

$$\left(1 + \frac{l(m+1)}{2} \frac{t_0}{t_a}\right) w_0$$

where l is the height of the precedence tree.

In section 2 we shall show how McNaughton's algorithm for the case $t_0 = 0$ can be adapted to get a fast algorithm, S, to schedule n independent tasks on m identical processors with identical memory size such that:

$$w_S \le t_a + \frac{(m-1)}{m} t_0$$

This new algorithm may be used in place of algorithm SU to schedule tree precedence task systems in the algorithm of [12]. The resulting algorithm produces better schedules. In this section, we also show that the worst case bound of $t_a + (m-1)t_0/m$ cannot be improved upon. In section 3, we consider identical processors with different memory size and finally, in section 4, we consider the case of uniform processors with the same memory size.

2. Identical Processors With The Same Memory Size

Mc Naughton's algorithm to construct a minimum length schedule for the case $t_0 = 0$ proceeds by first computing the schedule length f as below:

$$f = \max \{ \max_i \{ t_i \}, \sum_i t_i / m \}$$

The n tasks are now scheduled, in any order, by first using up all of processor 1 (P1), then all of P2, then all of P3, etc. until all n tasks have been scheduled. If when scheduling a task on Pi we discover that it cannot complete by f, then the remainder is assigned to Pi+1 starting at 0.

When $t_0 \neq 0$, we compute w_S as below:

$$w_s = \max \{ \max_i \{ (t_0 + t_i \}, (\sum_i (t_0 + t_i) + (m-1)t_0)/m \}$$

A modified version of Mc Naughton's algorithm is used to obtain a schedule of length at most w_S. The tasks are scheduled using algorithm S of Figure 4.

Note that it is possible for algorithm S to generate schedules that are shorter than w_S by upto $(m-1)t_0/m$. Theorem 1 establishes that algorithm S always succeeds in generating a valid schedule.

Theorem 1: Algorithm S always generates a schedule of length at most w_S.
Proof: We first observe that there are three points in the algorithm where a task might be scheduled. If it is scheduled at the point labeled 1:, it's scheduling satisfies criteria 1 and 2 stated earlier for valid schedules.

```
procedure S
i := 1; {task number}
j := 1; {proccssor numbcr}
q := m; {last available processor}
time_remaining := $w_S$; {remaining time on processor j}
for i := 1 to n do
    if $t_0 + t_i \leq$ time_remaining
    then begin
      1: Schedule task i on processor j for $t_0 + t_i$ time
         beginning at time $w_S$ - time_remaining;
         timc_rcmaining := timc_rcmaining - $t_0 - t_i$;
       if time_remaining $\leq t_0$
       then begin
        2:  j := j+1;
            time_remaining := $w_S$;
            end;
          end
    else begin
       if $w_S - t_0 - t_i \leq t_0$
       then begin
        3: Schedule task i on processor q from 0 to $t_0 + t_i$;
        4: q := q - 1;
             end
        else begin
        5: Schedule task i on processor j from $w_S$ − time_remaining
           to $w_S$ and on processor j+1 from 0 to $2t_0 + t_i$ - time_remaining;
           timc_rcmaining := $w_S$ + time_remaining $- 2t_0 - t_i$;
           j := j+1;
               end;

end; {of S}
```

Figure 4 Algorithm to schedule identical processors.

If task i is schedulcd at point 3:, thcn $t_0 + t_i \leq w_S$ by dcfinition of w_S. Again, the scheduling of task i is done in a valid way without increasing the schedule length beyond w_S.

Point 5: is the only place where a task may be scheduled with a preemption. We must show that the scheduling of the two subtasks that task i is divided into does not overlap. The sum of the task times for the two subtasks is $2t_0 + t_i$. This quantity cannot exceed w_S because if it did, then $w_S - t_0 - t_i < t_0$ and task i will be scheduled at point 3:. Hence, the two subtasks of task i do not overlap.

Finally, we need to show that by the time j exceeds q, all tasks have been scheduled. If this is not the case, then the schedule generated either uses more than m processors or has assigned more that one task for processing in the same time slot on some of the processors. Let i' be the first value of i when an attempt is made to schedule a task on a processor that has already been used (this processor would have been used earlier by 3:) or on a processor with index j, j > m. Suppose that in the scheduling of the previous i'-1 tasks, j had been incremented k_1 times at 2: and q had been decremented k_2 times at 4:. This means that on k = $k_1 + k_2$ processors there are no premptions.

The total capacity utilized is $\sum_{1}^{i'-1}(t_0 + t_i) + pt_0$, where, p is the number of preemptions introduced. Since, $w_S \geq \{\sum_{1}^{n}(t_0 + t_i) + (m-1)t_0\}/\text{m}$, the idle capacity on all m processors together must be at least $\sum_{i'}^{n}(t_0 + t_i) + (m-p-1)t_0 \geq (m-p)t_0 + t_{i'}$.

If j = q when i = i', then time_remaining on processor j is less than $t_0 + t_{i'}$ and the remaining processors have at most t_0 idle time each. In fact, the total idle time on the remaining processors is no more than kt_0. Hence the total idle time on the m processors is less than $kt_0 + t_0 + t_{i'}$. However, the number of preemptions in this case is m-k-1. So, the remaining capacity must be at least (k+1)$t_0 + t_{i'}$, a contradiction.

If j > q when i = i', then the total idle time on the m processors is at most kt_0. But, p=m-k and the available capacity must be at least k$t_0 + t_{i'}$.

Hence the algorithm always generates a valid schedule with length at most w_S ▪

Examining the definition of w_S, we see that $w_S \leq t_a + (m-1)t_0/m$. Our next theorem establishes that we cannot get a better bound on w_S.

Theorem 2: For every m, there exist task sets for which the minimum length schedule is of length $t_a + (m-1)t_0/m$.

Proof: First consider the case m = 2, $t_0 = 1$, $t_1 = t_2 = t_3 = 5$. One may readily verify that if no preemptions are allowed, then there is no schedule with length less than 12. If one or more preemptions are allowed, then there is no schedule with length less than $t_a + t_0/2 = 9.5$. This example generalizes to the case of m processors. Simply consider m+1 tasks of length 5 and $t_0 = 1$. ▪

We note that algorithm S is substantially simpler than the algorithm proposed in [12]. In fact, it can be trivially implemented in hardware, thereby virtually eliminating the scheduling overhead. For m = 2, the bound on w_S is the same as that on the algorithm of [12]. For other values of m, our bound is better by an additive amount of (m-1)(1/2-1/m). Also, our algorithm may be substituted into the algorithm suggested in [12] for tree precedence tasks. The resulting algorithm will have an improved performance. Since the minimum schedule length, w_0, is at least t_a, we obtain the relation $w_S \leq w_0 + (m-1)t_0/m$.

3. Identical Processors With Different Memory Size

Our heuristic algorithm for this case is based on the algorithm suggested by Kafura and Shen [5]. As remarked earlier, this algorithm generates optimal schedules when $t_0 = 0$ and it runs in O(nlogm) time. Assume that $\mu_i \geq \mu_{i+1}$, 1≤i<m. Let $B_i = \{j \mid \mu_{i+1} < u_j \leq \mu_i\}$, 1≤i<m and $B_m = \{j \mid u_j \leq \mu_m\}$. Let $F_i = \bigcup_{1}^{i} B_j$, 1≤i≤m and let $X_i = \sum_{j \in F_i} t_j$. Define f as below:

$$f = \max \{ \max_i \{ t_i \}, \max_i \{ X_i / i \} \}$$

The Kafura-Shen algorithm generates schedules of length f by scheduling first all jobs in B_1, then all in B_2, and so on. When tasks from B_i are being considered, processors 1 through i are available. The scheduling is done using McMaughton's scheme. It is not too difficult to see that when $t_0 \neq 0$, this strategy can be adapted in the same way as we adapted McNaughton's algorithm in section 2. The w_S to use now is given below:

$$w_S = \max\{\max_i\{t_i + t_0\}, \max_i\{(Y_i + (i-1)t_0)/i\}\}$$

where $Y_i = \sum_{j \in F_i} (t_j + t_0)$.

The correctness of the scheduling method may be established as in section 2.

4. Uniform Processors With Equal Memory Size

Assume that the processors are ordered by speed. I.e., $s_i \geq s_{i+1}$, $1 \leq i < m$. Let T_i and S_i, $1 \leq i \leq m$ be as defined below:

T_i = sum of the longest i task times + it_0, $1 \leq i < m$.

T_m = sum of the n task times + nt_0

$$S_i = \sum_{j=1}^{i} s_i, \quad 1 \leq i \leq m$$

When $t_0 = 0$, a minimum length schedule can be obtained in O(n + mlogm) time [3]. The algorithm of [3] begins by computing the minimum schedule length, f, using the formula:

$$f = \max_{1 \leq k \leq m}\{T_k / S_k\}$$

Since the algorithm of [3] generates schedules that have no more than 2(m-1) preemptions, one might conjecture that in the face of overheads of $t_0 > 0$ per preemption, the schedule length need increase to no more than w_U as given below:

$$w_U = \max_{1 \le i \le m} \{ (T_i + 2(i-1)t_0) / S_i \}$$

Establishing the validity of the above conjecture is quite a bit harder than establishing the validity of the bound w_S for identical processors. Like the algorithm of [3] for the case when $t_0 = 0$, our algorithm here will use 4 scheduling rules. However, the condition for applying each and the rules themselves are somewhat different. The n tasks shall be scheduled one-by-one. The schedule for any given task will be obtained by using exactly one of the 4 rules.

Let us introduce some terminology first. Processor j has *idle time* if there is some time between 0 and w_U during which no task has been assigned to it. The interval [a,b] constitutes a *block* of idle time on processor j iff this processor is idle throughout this interval. A block [a,b] of idle time on processor j is a *usable block* iff $(b-a)s_j > t_0$. A set of processors with nonoverlapping usable blocks is called a *usable processor system* (UPS).

A three processor system with idle times is shown in Figure 5(a). The heavy lines represent nonusable idle blocks while the light lines represent usable blocks. Note that there is no overlap amongst the usable blocks. This represents a UPS even though some usable blocks overlap with some nonusable blocks. A UPS will be drawn as in Figure 5(b). In this figure, only the usable blocks are shown. Observe that unlike the DPS of [3], a UPS is not required to consist of a continuous block of idle time from 0 to w_U.

Let us assume that $t_1 \ge t_2 \ge \cdots \ge t_{m-1} \ge t_j$, $j \ge m$. Task i is the ith task to be scheduled. We shall use k to denote the next task to be scheduled. Initially, k=1. I(k) will denote the set of processors used in the scheduling of tasks 1, 2, ..., k-1. Initially, I(k) = {1}. idle_time(k) denotes the total amount of processing capacity available in the usable blocks of I(k) (i.e., sum of the block length and speed products). NP(k) is the number of preemptions in the schedule

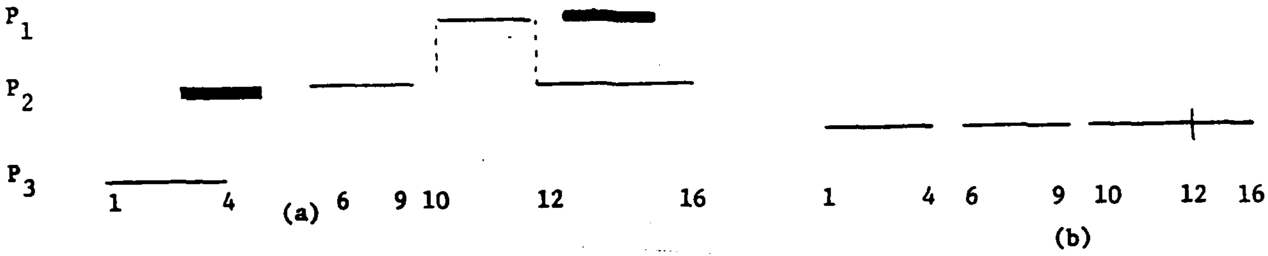


Figure 5 A UPS

constructed for tasks 1, 2, ..., k-1; H(k) is the number of usable blocks in I(k); and A(k) is the number of unusable idle blocks in I(k). Note that each unusable block represents atmost t_0 units of processing.

When task k is to be scheduled, we determine which of conditions C1 - C4 (given below) holds and use the appropriate scheduling rule. Informally, these four conditions are:

C1: Task k can be scheduled on the usable blocks of I(k) in such a way that no usable blocks remain.

C2: There isn't enough usable capacity in I(k) to complete task k.

C3: The usable processing capacity in I(k) is enough to complete task k. However, the usable capacity left following the scheduling of this task will exceed t_0.

C4: (k=m) or there is enough usable capacity in I(k) as well as on each of the processors not in I(k) to complete task k.

These conditions are specified more formally later. They are tested for in the order C4, C1, C2, and C3. Once C4 holds, rule R4 takes over and schedules all remaining tasks. For every k such that task k-1 is scheduled using one of rules R1-R3, the following will be true:

1. I(k) is a UPS.

2. $|I(k)| = k$

3. $NP(k) + H(k) + A(k) - 1 \leq 2(k-1)$.

When k=1, NP(k)=0, H(k)=1, A(k)=0 and we see that 1-3 above are true.

The four scheduling rules together with their associated conditions are given below.

Rule R1

Condition C1: $H(k)t_0 + t_k \leq idle\_time(k) \leq (H(k)+1)t_0 + t_k$

Task k is scheduled in the H(k) usable blocks of I(k). This scheduling may leave behind an unusable block of size upto t_0 (Figure 6). Let j be the index of the fastest processor not in I(k). Such a j must exist as $|I(k)| = k < m$. Define I(k+1) to be I(k) ∪ {j}. The time on processor j from 0 to w_{ij} constitutes the only usable block of I(k+1). We see that NP(k+1) = NP(k) + H(k) - 1, H(k+1) = 1, and A(k+1) ≤ A(k) + 1. Hence,

$$\begin{aligned} NP(k+1) + H(k+1) + A(k+1) - 1 &\leq NP(k) + H(k) - 1 + 1 + A(k) + 1 - 1 \\ &= NP(k) + H(k) + A(k) - 1 + 1 \\ &\leq 2(k-1) + 1 \\ &< 2k. \end{aligned}$$

Also, I(k+1) is a UPS and $|I(k+1)| = k+1$.

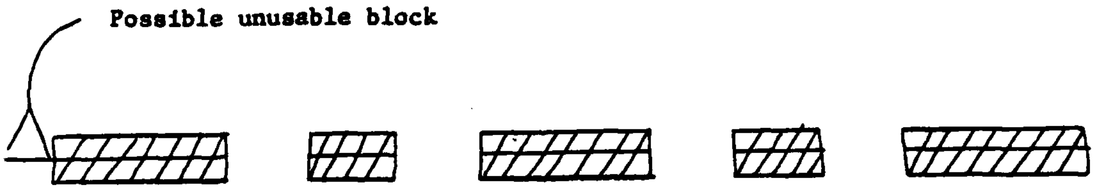


Figure 6 Scheduling with rule R1

Rule R2

Condition C2: idle_time(k) < $H(k)t_0 + t_k$

At this time, there isn't enough usable processing capacity in I(k) to schedule task k. Let I(k) = $\{1, 2, \ldots, j, i_1, i_2, \ldots\ i_{k-j}\}$, where $j+1 \not\in S = \{i_1, i_2, \ldots\}$. If $S = \phi$, then j=k and idle_time(k) $\geq (T_k + 2(k-1)t_0) - (T_{k-1} + NP(k)t_0 + A(k)t_0) \geq t_k+t_0+2(k-1)t_0+(H(k)-1)t_0-2(k-1)t_0 = t_k+H(k)t_0$ (recall k < m when rule R2 is used). This contradicts condition C2. So, $S \neq \phi$. Processors in S are introduced into I by rule R3. From the way this rule selects a processor for inclusion and the fact $t_1 \geq t_2 \geq \cdots \geq t_k$, it follows that $t_k + t_0 \leq w_u s_{j+1}$.

Let I(k+1) = I(k) ∪ {j+1}. So, |I(k+1)| = k+1. Index the usable blocks of I(k) 1 through H(k). Let τ_i denote the start of the ith usable block. Let Δ_i be the processing capacity of the ith usable block (i.e., the product of block length and processor speed). Assume that the usable blocks have been indexed such that $\tau_i > \tau_{i+1}$, 1≤i<H(k) (Figure 7). Let $\tau_0 = w_u$. Find the least i, i≥0, such that one of the following is true:

a) $(i+1)t_0 + t_k \leq \sum_{p=1}^{i} \Delta_p + \tau_i s_{j+1} \leq (i+2)t_0 + t_k$

b) $\sum_{p=1}^{i} \Delta_p + \tau_i s_{j+1} < (i+1)t_0 + t_k$

c) $i = H(k)$

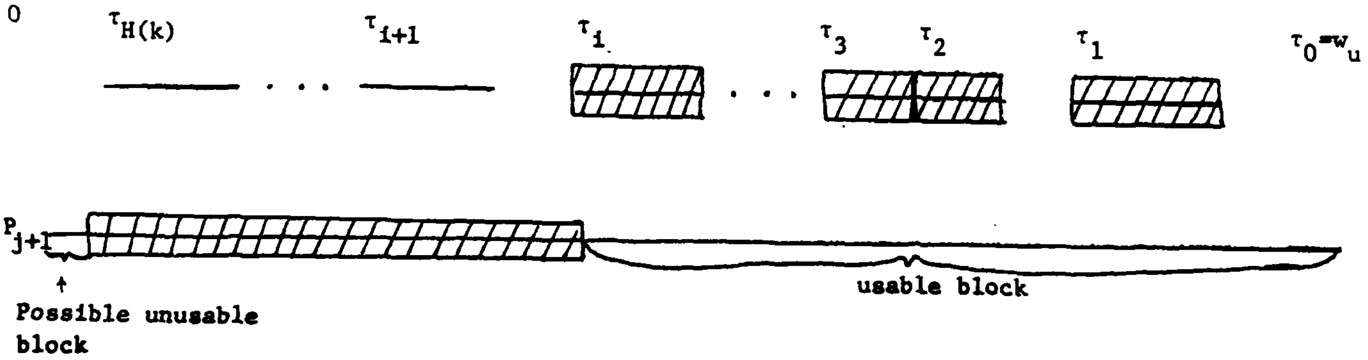


Figure 7 Scheduling with rule R2(a)

Clearly, such an i exists. The scheduling of task k depends on which of the above conditions holds for this i. If more than one of the above hold for this least i, then the first of them that holds determines the way to schedule task k.

Case (a) holds

Schedule task k to completely use up the usable blocks 1 through i. Schedule the remainder of this task on processor j+1 so as to finish at τ_i (Figure 7). The remaining usable block (if any) on j+1 begins at τ_i and ends at w_{ij}. If i=0, then there is no usable idle time left on processor j+1. If i>0, then it follows that the processing capacity of j+1 from τ_i to w_{ij} is greater than t_0. Also, note that the scheduling of task k might create an unusable idle block of capacity at most t_0 starting at 0 on j+1. It is not too difficult to see that NP(k+1) = NP(k) + i, H(k+1) ≤ H(k) - i + 1, and A(k+1) ≤ A(k) + 1. Hence,

NP(k+1) + H(k+1) + A(k+1) - 1
≤ NP(k) + i + H(k) - i + 1 + A(k) + 1 - 1
≤ 2(k-1) + 2
= 2k.

We also note that I(k+1) is a UPS.

Case (b) holds

Now, i>0 as $\tau_0 s_{j+1} \geq t_k + t_0$. So,

(1) $\sum_{p=1}^{i-1} \Delta_p + \tau_{i-1} s_{j+1} > i t_0 + t_k$

and

(2) $\sum_{p=1}^{i} \Delta_p + \tau_i s_{j+1} < (i+1) t_0 + t_k$

Also, observe that:

$$\sum_{p=1}^{i-1} \Delta_p + \tau_{i-1} s_{j+1} > (i+1) t_0 + t_k$$

as otherwise case (a) occurs for i-1.

This time we assign task k so as to use up the usable blocks 1 through i-1 (Figure 8). Let β be the end of the ith usable block and let s = $\Delta_i / (\beta - \tau_i)$. Note that it is quite possible that $\beta < \tau_{i-1}$ (of course, it is not possible for β to be greater than τ_{i-1}). Let $\delta = (i t_0 + t_k - \sum_{p=1}^{i-1} \Delta_p) / s_{j+1})$. From (1) it is evident that $\delta < \tau_{i-1}$. If $\delta \geq \beta - t_0 / s_{j+1}$, then schedule the remainder of task k on j+1 from 0 to δ. If in addition, $\delta < \beta$, then designate the time from δ to β on j+1 unusable. The only usable block on j+1 begins at max{β, δ} and ends at w_{ij}. We see that when task k is completed in this way, NP(k+1) = NP(k) + i - 1, H(k+1) = H(k) + 1 - (i-1), and A(k+1) ≤ A(k) + 1. So, NP(k+1) + H(k+1) + A(k+1) - 1 ≤ 2(k-1) + 2 = 2k.

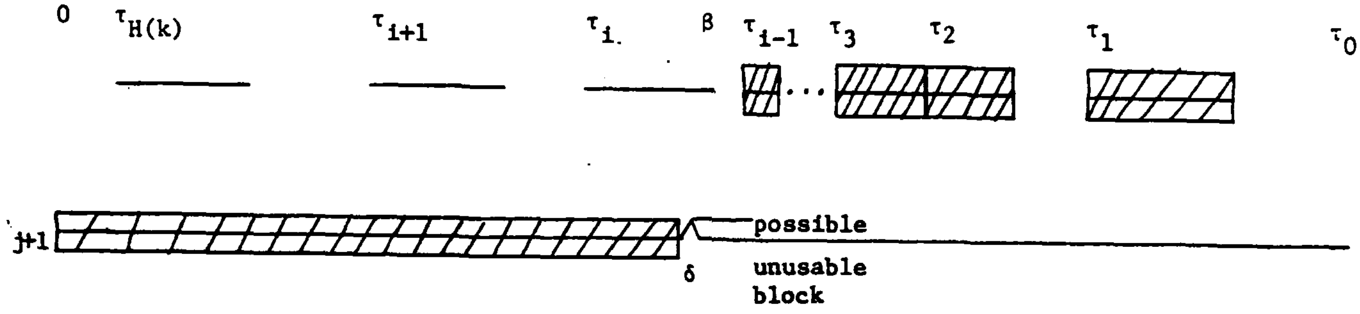


Figure 8 Scheduling with rule R2(b)

If $\delta < \beta - t_0/s_{j+1}$, then $\beta s_{j+1} > (i+1)t_0 + t_k - \sum_{p=1}^{i-1}\Delta_p$. From (2), we know that $(\beta-\tau_i)s + \tau_i s_{j+1} < (i+1)t_0 + t_k - \sum_{p=1}^{i-1}\Delta_p$. So, there is a γ, $\tau_i < \gamma < \beta$ such that $\gamma s_{j+1} + (\beta-\gamma)s = (i+1)t_0 + t_k - \sum_{p=1}^{i-1}\Delta_p$. The remainder of task k is scheduled on j+1 from 0 to γ and in the usable block Δ_i from γ to β (Figure 9). The idle time on j+1 from γ to w_{ij} forms a usable block. If the remaining idle capacity in Δ_i is no more than t_0, then an unusable block is created here. So, NP(k+1) = NP(k) + i, and H(k+1) + A(k+1) ≤ H(k) + A(k) - i + 2. Hence, NP(k+1) + H(k+1) + A(k+1) - 1 ≤ 2k. I(k+1) is readily seen to be a UPS.

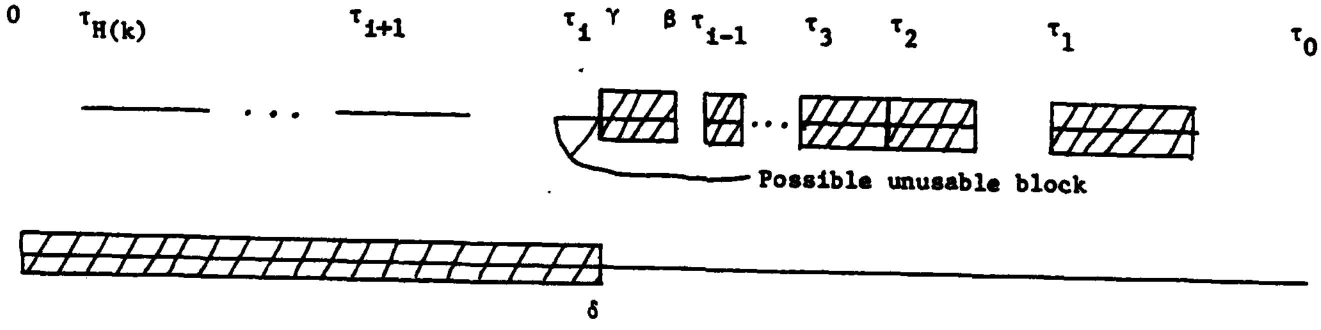


Figure 9 Scheduling with rule R2(b)

Case (c) holds

It is the case that $\sum_{p=1}^{H(k)} \Delta_p + \tau_{H(k)} s_{j+1} > (H(k)+1)t_0 + t_k$. Let $\gamma = ((H(k)+1)t_0 + t_k - \sum_{p=1}^{H(k)} \Delta_p)/s_{j+1}$. Schedule task k to use up the usable blocks of I(k) and on j+1 form 0 to γ (Figure 10). It is clear that $\gamma < \tau_{H(k)}$. I(k+1) has only one usable block. It is on processor j+1 from γ to w_g. NP(k+1) = NP(k) + H(k), and A(k+1) = A(k). So, NP(k+1) + H(k+1) + A(k+1) - 1 < 2k.

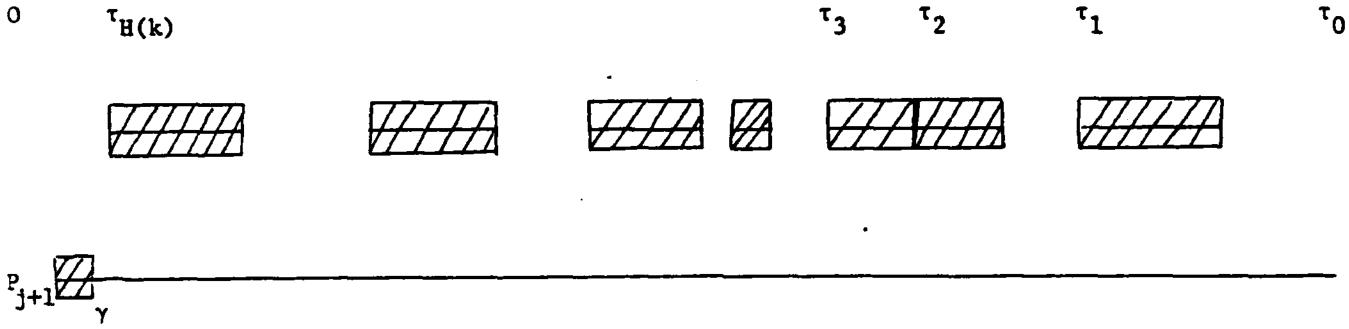


Figure 10 Scheduling with rule R2(c)

Rule R3

Condition C3: idle_time(k) > (H(k)+1)t_0 + t_k

Let q be the smallest value of r such that $r \notin$ I(k) and $t_k + t_0 > w_g s_r$. Such a q must exist as otherwise C4 also holds and is given priority over this rule. Let τ_i, Δ_i, 1≤i≤H(k) be as in rule R2. Let I(k+1) = I(k) ∪ {q}. We first see that |I(k+1)| = k+1.

Find the largest i, i<H(k), for which one of the following true:

a) $(i+1)t_0 + t_k \le \sum_{p=1}^{i} \Delta_p + \tau_i s_q \le (i+2)t_0 + t_k$

b) $\sum_{p=1}^{i} \Delta_p + \tau_i s_q < (i+1)t_0 + t_k$

c) $i = 0$

Clearly, such an i exists. The scheduling of task k depends on which of the above conditions holds for this i. If more than one of the above hold for this largest i, then the first that holds determines the way to schedule task k.

Case (a) holds

Schedule as in case (a) of Rule R2.

Case (b) holds

We have the following inequalities:

$$\tau_{i+1}s_q + \sum_{p=1}^{i+1}\Delta_p > (i+2)t_0 + t_k$$

and

$$\tau_i s_q + \sum_{p=1}^{i}\Delta_p < (i+1)t_0 + t_k$$

Let β be the end of the usable block Δ_{i+1}. From the last inequality and the relation $\beta \le \tau_i$, it follows that:

$$\beta s_q + \sum_{p=1}^{i}\Delta_p < (i+2)t_0 + t_k$$

Hence, there exists γ, $\tau_{i+1} < \gamma < \beta$, such that

$$\gamma s_q + (\beta-\gamma)s + \sum_{p=1}^{i}\Delta_p = (i+2)t_0 + t_k$$

where $s = \Delta_{i+1}/(\beta-\tau_{i+1})$. Task k is scheduled on processor q from 0 to γ, on the usable block Δ_{i+1} from γ to β, and on the whole of the usable blocks indexed 1 through i. The idle time on processor q from γ to w_q may or may not form a usable block. Further, the capacity left on Δ_{i+1} may also be unusable. Regardless of the outcome for the remaining capacity on q and the i+1th block, we have NP(k+1) = NP(k) + i + 1, and H(k+1) + A(k+1) ≤ H(k) + A(k) + 1 - i. So, NP(k+1) + H(k+1) + A(k+1) - 1 ≤ 2k.

case (c) holds

When H(k) = 1, (3) follows from C3. When H(k) > 1, (3) follows from (a) and (b) with i=1.

(3) $\Delta_1 + \tau_1 s_q > 2t_0 + t_k$

Let β be the end of the interval Δ_1. From the choice of q and the relation $\beta \leq wy$, we obtain:

(4) $2t_0 + t_k > t_0 + t_k > wys_q \geq \beta s_q$

From (3) and (4), it follows that there is a γ, $\tau_1 < \gamma < \beta$ such that:

$$\gamma s_q + (\beta-\gamma)\Delta_1 / (\beta-\tau_1) = 2t_0 + t_k$$

Schedule task k on processor q from 0 to γ and on Δ_1 from γ to β. The remaining idle time on Δ_1 and on q may or may not form usable blocks. Regardless of this, we have NP(k+1) = NP(k) + 1, H(k+1) + A(k+1) ≤ H(k) + A(k) + 1. So, NP(k+1) + H(k+1) + A(k+1) - 1 ≤ 2k.

Note Before moving on to rule R4, we should observe that the schedules generated by rules R2 and R3 may in fact assign task k for less than t_0 units of processing on some processors. This creates no problem as the schedule can be cleaned up in the end; eliminating these assignments. Each such elimination reduces the number of preemptions by 1 and increases the value of A() by 1. So, the sum NP() + H() + A() - 1 is unchanged.

Rule R4

Condition C4: (k=m) or (C3 and $t_0 + t_k \leq wys_p$ for every p∉I(k))

If k=m, then the sum of the processing capacities in the usable blocks of I(m) is at least:

$$T_m + 2(m-1)t_0 - \sum_{i=1}^{m-1}(t_0+t_i) - (NP(m)+A(m))t_0$$

$$\geq T_m - \sum_{1}^{m-1}(t_0+t_i) + (H(m)-1)t_0$$

This is just enough to schedule the remaining n-m+1 tasks on the H(m) usable blocks of I(m) in the obvious way. At most H(m)-1 preemptions will be introduced and we have the idle capacity to handle this many additional overheads.

If k<m, then let $Q_1, Q_2, \ldots, Q_{m-k}$ be the processors not in I(k). Let q_i denote the speed of Q_i and assume that the processors have been ordered such that $q_i \leq q_{i+1}$, $1 \leq i < m-k$. We now schedule as many tasks as possible using the procedure given in Figure 11. This scheduling procedure is quite similar to McNaughton's [10]. We need to show that the preemptive scheduling done here does not cause an overlap. Let j be the index of the first task that is scheduled with an overlap. Let Q_p and Q_{p+1} be the processors on which it is scheduled. Let Δ be the amount of time it is assigned to Q_{p+1}. So, $\Delta q_{p+1} + (w_y - \Delta)q_p < 2t_0 + t_j$. Also, there must be an r, k≤r<min{m,j} and a v, 1≤v≤p, such that $2t_0 + t_r < w_y q_v$. If this is not the case, then $t_0 + t_l \le w_y q_{l-k+1} \le 2t_0 + t_l$, k≤$l$<min{m,j}. So, tasks k, k+1, ..., min{m,j}-1 are scheduled to use up all of $Q_1, Q_2, \ldots, Q_{\min\{m,j\}-1}$ respectively. If j<m, then task j is to be scheduled by the **then** clause of Figure 11 and no preemption occurs. If j≥m, then p>m-k and task j is not scheduled by Figure 11. So, we may assume that r and v as described above exist. Now, since $t_j \le t_r$ and $q_{p+1} \ge q_p \ge q_v$, it must be that $2t_0 + t_j < \Delta q_{p+1} + (w_y - \Delta)q_p$. A contradiction. Hence, no task is scheduled with overlap.

Let numpi be the total number of preemptions and idle slots of size at most t_0 that are introduced. We see that if no usable block remains on Q_{m-k}, then numpi ≤ m-k. Otherwise, numpi ≤ m-k-1.

If j>n when this procedure terminates, then all tasks have been scheduled and we need go no further. If j≤n, then it is necessary to schedule some tasks in the usable blocks of I(k). If the idle capacity left on Q_{m-k} is no more than t_0, then the usable capacity in I(k) is at least $T_m + 2(m-1)t_0 - \sum_{i=1}^{j-1}(t_i + t_0) - (\mathrm{NP(k)} + \mathrm{A(k)} + \mathrm{numpi})t_0 \ge \sum_{i=j}^{n}(t_i + t_0) + 2(m-1)t_0 - \{2(k-1) + 1 - H(k) + m - k\}t_0 = \sum_{i=j}^{n}(t_i + t_0) + (m-k+H(k)-1)t_0$. This is enough capacity to process the remaining tasks in a straightforward way.

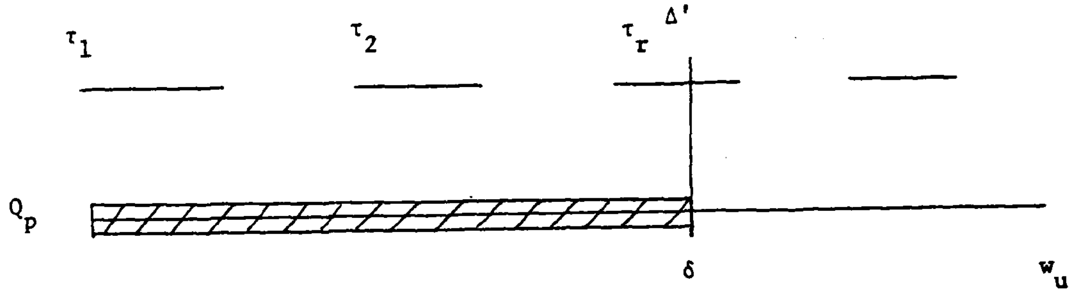
The final case to consider is when the idle capacity left on Q_{m-k} exceeds t_0. Let the idle time on Q_{m-k} begin at δ and go upto w_y. The capacity associated with this time is less than $t_0 + t_j$. If there is no overlap between the idle time on Q_{m-k} and the usable blocks of I(k), then we may schedule the remaining tasks on the H(k)+1 usable blocks in a straightforward way introducing at most H(k) additional preemptions and idle slots of capacity at most t_0 each. We may verify that enough capacity exists for this. So, assume that there is some overlap. We

```
p:=1; j:=k; idle_time := $w_{ij}q_1$;
rcpcat
  if $t_0 + t_j \le$ idie_time
  then begin
          schedule j on $Q_p$;
          idle_time := idle_time - $t_0 - t_j$;
          if idle_time $\le t_0$
          then begin
                  p := p+1;
                  idlc_timc := $w_{ij}q_p$;
               end;
       end
  else begin
          if p = m-k then exit;
          schedule j on $Q_p$  upto $w_{ij}$ and on $Q_{p+1}$ beginning
          at 0.  This requires exactly one preemption.
          idle_time := $w_{ij}q_{p+1}$ + idle_time $- 2t_0 - t_j$;
          p := p+1;
       end;
  j := j+1;
until j>n or p>m-k;
```

Figure 11

have the situation of Figure 12. For convenience, we have numbered the blocks lcft to right in this figurc. r is thc highcst indcx such that block i of I(k) has some overlap with the idle time on Q_{m-k}. Clearly, r≥1. Let the capacity of the ith block be Δ_i and let s = q_{m-k}.

If $\sum_{1}^{r-1} \Delta_p + (w_{ij} - \delta)s \ge rt_0 + t_j$, then schedule task j to use up all of Q_{m-k} and as much of $\Delta_1, \Delta_2, \ldots, \Delta_{r-1}$ as needed to complete task j. One may easily show that there is enough capacity left to complete the remaining tasks by scheduling them as for the case when k=m.



Figurc 12

If $\sum_{1}^{r-1}\Delta_p + \Delta' + (w_U-\delta)s \geq (r+1)t_0 + t_j$, where Δ' is the capacity available in the rth block from τ_r upto δ, then again scheduie task j to use up all of Q_{m-k}, all of Δ_i, 1≤i<r, and the appropriate needed fraction of Δ_r. Once again, we may verify that there is enough remaining capacity in I(k) to complete the remaining tasks by scheduling them as we did for the case k=m.

Otherwise, from C4 and k<m, we see that there is an i≥r for which $\sum_{p=1}^{i}\Delta_p + (w_U-\beta_i)s \geq (i+1)t_0 + t_j$, ($\beta_i$ is the end of the ith usable block). Find the least i for which this is true. It follows that $\sum_{p=1}^{i-1}\Delta_p + (w_U-\tau_i)s < (i+1)t_0 + t_j$. Hence, there is a γ, $\tau_i < \gamma \leq \beta_i$, such that task j can be completed by scheduling it on all of Δ_p, 1≤p<i, on the ith usable block from τ_i to γ, and on Q_{m-k} from γ to w_U. One may verify that the remaining capacity is enough to complete the remaining tasks. Since all remaining usable blocks are nonoverlapping, the remaining tasks are easily scheduled.

Complexity

The scheduling algorithm described above can be implemented in O(n+mlogm) time. mlogm time is needed to order the processors by speed and n+mlogm time is needed to obtain the m longest tasks in sorted order.

5. Conclusions

We have shown that it is possible to efficiently generate "good" schedules for various systems of processors in the face of preemptive overheads. For the case of identical processors with or without different memory size the schedules generated are within $(m\text{-}1)t_0/m$ of the optimal schedules. When processors have different speeds but equal memory size the schedules generated by our algorithm are within $\max_{1 \le i \le m}\{2(i-1)t_0/S_i\}$ of the optimal schedule length. Our result for identical processors represents an improvement over the results obtained in [12].

6. References

[1] J. Bruno and P. Downey, "Complexity of task sequencing with deadlines, set-up times, and changeover costs," *SIAM Computing*, Nov. 1978, 393-404.

[2] J. Bruno, J,. Jones, and K. So, "Deterministic scheduling with pipelined processors," *IEEE Trans. On Computers*, April 1980, 308-316.

[3] T.Gonzalez and S.Sahni, "Preemptive scheduling of uniform processor systems", *JACM*, Jan. 1978, 92-101.

[4] M. Garey and D. Johnson, *Computers and intractability*, W.H. Freeman and Co., 1979.

[5] D.Kafura and V.Shen, "Task scheduling on a multiprocessor system with independent memories", *SIAM Computing*, March 1977, 167-187.

[6] P.Kogge, *The architecture of pipelined computers*, McGraw Hill Book Co., New York, 1981.

[7] T.Lai and S.Sahni, "Preemptive scheduling of uniform processors with memory", Technical Report, University of Minnesota, 1982.

[8] H.Li "Scheduling trees in parallel/pipelined processing environments," *IEEE Transactions on Computers*, Nov. 1977, 1101-1112.

[9] C.Martel, "Scheduling multiple processors with memory constraints," Proceedings 10th IMACS Congress, Aug. 1982.

[10] R.McNaughton, "Scheduling with deadlines and loss functions," *Manag. Sci.*, Oct. 1959, 1-12.

[11] C.Ramamoorthy, and H.Li, "Sequencing control in multifunctional pipeline systems", *Sagamore Computer Conference On Parallel Processing", 1975, 79-89.*

[12] S.Su and K.Hwang, "Multiple pipeline scheduling in vector supercomputers", ***1982 International Conference On Parallel Processing***, 226-234.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO. A132512

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Scheduling Supercomputers

5. TYPE OF REPORT & PERIOD COVERED
Technical Report February 1983

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Sartaj K. Sahni

8. CONTRACT OR GRANT NUMBER(s)
N00014-80-0650

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Computer Science Department
University of Minnesota
136 Lind Hall, 207 Church St. SE, Mpls, MN 55455

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Department of the Navy
Office of Naval Research
Arlington, VA 22217

12. REPORT DATE
February 1983

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Supercomputers, pipelines, asynchronous processors, scheduling, heuristics.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We develop good heuristics to schedule tasks on supercomputers. Supercomputers comprised of multiple pipelines as well as those comprised as asynchronous multiple processors are considered. In addition, we consider the case when different pipes or processors run at different speeds.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

END

DATE
FILMED

9 83

DTIC